Lab 10: Command line parameters

Goals

Demonstrate proficiency in the use of the switch construct and in processing parameter data passed to a program via the command line.

Background

It is often useful to pass arguments to a program via the command line. For example,

```
gcc -g -Wall -o p10 p10.c
```

In this case the C compiler, *gcc*, is being passed 6 different arguments.

```
0   gcc
1   -g
2   -Wall
3   -o
4   p10
5   p10.c
```

The *switch* statement

This *switch()* statement provides a slightly more civilized way to select from a large collection of possibly legal choices than the *if () else if() else if ()* approach. If the value of opchar matches one of the case definitions, the code immediately following the *case* clause wiill be executed.

Note that you *must* include the *break* statements shown. If you don't do that, execution will continue into the next case!!

```
switch (opchar)
{
case '+':
    answer = value1 + value2;
    break;

case '-':
    answer = value1 - value2;
    break;
:
: etc,
:
default:
    fprintf(stderr, "bad op code %c \n", opchar);

    return(-1);
}
```

The *default:* case is executed if an only if no other case matches. Here this means an unsupported operation code was read.

Printing command line arguments

When a program is started from the command line, the character strings (separated by spaces) comprising the program name and the remaining command line arguments are copied by the Operating System into the memory space of the new program and converted to standard NULL terminated character strings. Both *the total number of arguments* and an *array of character pointers to the individual character strings* are then passed to the main() function.

The count is conventially called *int argc* and the table *char *argv[].* These values can be accessed by the *main()* function as shown below.

```
#include <stdio.h>

int main(
int argc,     /* number of command line arguments    */
char *argv[]) /* array of addresses of the arguments */
{
   int ndx = 0;

   while (ndx < argc)
   {
      fprintf(stdout, "%s\n", argv[ndx]);
      ndx = ndx + 1;
   }
}
```

When the above program is given the name *printargs* and invoked as:

```
==> printargs -Wall -o hello -g myprog.c
```

Its output will be:

```
printargs
-Wall
-o
hello
-g
myprog.c
```

Processing numerical values with *sscanf()*

Suppose your mission is to write a program named *flag* whose function is to produce a .ppm image of a flag. Your program is to be invoked as:

```
flag width-of-flags-in-pixels
```

or for example:

```
flag 800
```

The value 800 is passed to your program as a 4 byte character string consisting of the ASCII encoding of the numerals 8, 0, and 0 and followed by the binary 0 byte signalling end of string.

The *sscanf()* function works in a manner that is analogous to *fscanf()* and supports the same set of format codes (%d, %f, %lf, %c, %s, etc). The only difference is that *sscanf()* reads from a NULL terminated string that is *already present in memory* instead of a file.

Therefore, to read the width into an integer variable use:

```
howmany = sscanf(argv[1], "%d", &width);
```

This will work because *argv[1]* is a pointer to the string holding "800". If it were necessary to read *width* and *depth* values, as in:

flag 800 600

two calls to *sscanf()* are required because each parameter is stored in a separate string:

```
howmany  = sscanf(argv[1], "%d", &width);
howmany += sscanf(argv[2], "%d", &depth);
```

Exercise: Why would it NOT work to use strcat to catenate the two strings and then do:

```
howmany  = sscanf(&catstring[0], "%d %d", &width, &depth);
```

Assignment:

In this assignment you will create programs called part1.c and part2.c that will consist of a single main(int argc, char *argv[]) function. This function will perform *double precision* floating point calculations on command line parameters and print the result. You must support four operations: +, -, x, /. NOTE: *use x not * for multiply!!!*

Exercise: What happens if you do insert * instead of *x* in the command line??

Part 1:

For part 1 of the assignment your calculator must support exactly two operands and one operator. It must use *sscanf()* with the %lf format code to acquire the double precision floating point values and may use *sscanf()* with the %c format code to acquire the operator. Use the *switch ()* statement to distinguish among the four cases. Print the result of the floating point operation to the *standard output* using the %12.3lf format.

Examples:

```
a.out 12 x 4
      48.000
a.out 5 / 2.0
       2.500
a.out 5 - 6.0
      -1.000
```

Part 2:

When you complete part1.c move on to part2.c. For part 2 the command line will contain an arbitrary number of parameters but they will always have the following organization in which the first and last parameters are floating point numbers and all numbers are separated by exactly one operator.

a.out fp-num operator fp-num operator fp-num operator fp-num

The expression should be evaluated in *strict left to right order*.

```
a.out 4 + 2 x 3 - 6
      12.000

a.out 1 + 2 + 3 + 4 + 5 + 6 - 6 - 5 - 4 - 3 - 2 - 1
       0.000
```

Turn In Work

Show your TA that you completed the assignment. Then turn in your part1.c and part2.c programs using the command:

sendlab.101.*section_number* 10 part1.c part2.c